CHAPTER 2
Emerging JavaScript

Since its release in 1995, JavaScript has gone through many changes. At first, it made adding interactive elements to web pages much simpler. Then it got more robust with DHTML and AJAX. Now with Node.js, JavaScript has become a language that is used to build full-stack applications. The committee that is and has been in charge of shepherding the changes to JavaScript is the ECMA, the European Computer Manufacture Association.

Changes to the language are community driven. They originate from proposals that community members write. Anyone can submit a proposal to the ECMA committee. The responsibility of the ECMA committee is to manage and prioritize these proposals to decide what is included each spec. Proposals are taken through clearly defined stages. Stage-0 represents the newest proposals up through Stage-4 which represents the finished proposals.

The most recent version of the specification was approved in June 2015 [1] and is called by many names: ECMAScript 6, ES6, ES2015, Harmony, or ESNext. Based on current plans, new specs will be released on a yearly cycle. For 2016, the release will be relatively small [2], but it already looks like ES2017 will include quite a few useful features. We'll be using many of these new features in the book and will opt to use emerging JavaScript whenever possible.

Many of these features are already supported by the newest browsers. We will also be covering how to convert your code from emerging JavaScript syntax to ES5 syntax that will work today in most all of the browsers. The kangax compatibility table is a

1 "ECMAScript 2015 Has Been Released", InfoQ, June 17, 2015 https://www.infoq.com/news/2015/06/ecmascript-2015-es6

2 ES2016 Spec Info: https://tc39.github.io/ecma262/2016/

great place to stay informed about the latest JavaScript features and their varying degrees of support by browsers.

In this chapter, we will show you all of the emerging JavaScript that we'll be using throughout the book. If you haven't made the switch to the latest syntax yet, then now will be a good time to get started. If you are already comfortable with ESNext language features, you can go ahead and skip to the next chapter.

Declaring Variables in ES6

Const

ES6 introduced constants, and they are already supported in most browsers. A constant is a variable that cannot be changed. It is the same concept as constants in other language.

Before constants, all we had were variables, and variables could be overwritten.

```
var pizza = true
pizza = false
console.log(pizza) // false
```

We cannot reset the value of a constant variable, and it will generate a console error if we try to overwrite the value.

```
const pizza = true
pizza = false
```

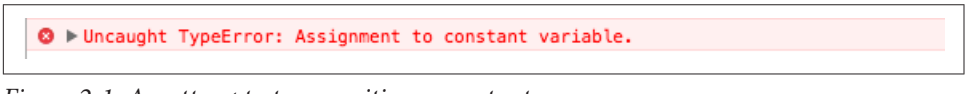


Figure 2-1. An attempt at overwriting a constant

Let

JavaScript now has lexical variable scoping. In JavaScript, we create code blocks with curly brackets. With functions, these curly brackets block off the scope of variables. On the other hand, think about if/else statements. If you come from other languages, you might assume that these blocks would also block variable scope. This is not the case.

If a variable is created inside of an if/else block, that variable is not scoped to the block.

```
var topic = "JavaScript"

if (topic) {
```

```
  var topic = "React"
  console.log('block', topic)      // block React
}

console.log('global', topic)       // global React
```

The topic variable inside the if block resets the value of topic.

With the let keyword, we can scope a variable to any code block. Using let protects the value of the global variable.

```
var topic = "JavaScript"

if (topic) {
  let topic = "React"
  console.log('block', topic)      // React
}

console.log('global', topic)       // JavaScript
```

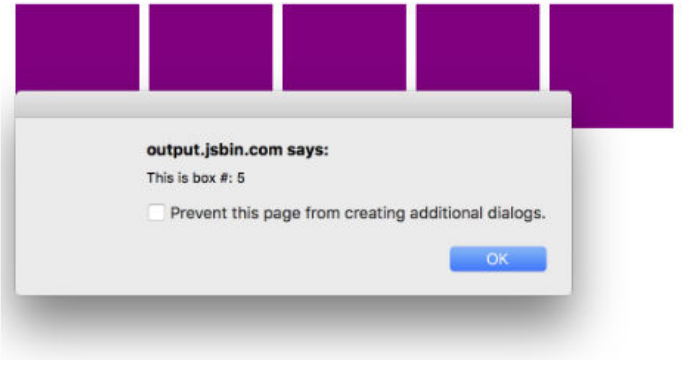
The value of topic is not reset outside of the block.

Another area where curly brackets don't block off a variable's scope is in for-loops.

```
var div,
    container = document.getElementById('container')

for (var i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
      alert('This is box #' + i)
   }
  container.appendChild(div)
}
```

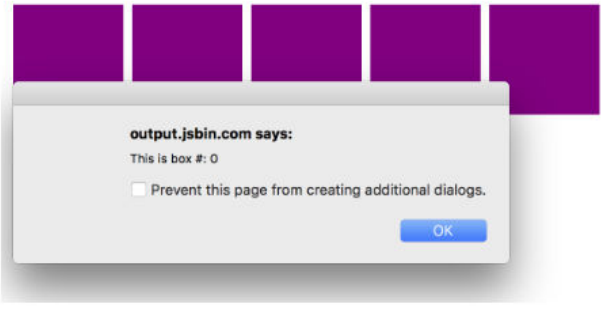
In this for-loop, we create 5 divs to appear within a container. Each div is assigned an onclick handler that alerts a message to displays the index. Declaring i in the for loop creates a global variable named i, and then iterates it until it reaches 5. When you click on any of these boxes, the alert says that i is equal to 5 for all divs, because the current value for global i is 5.



Output

Declaring the loop counter i with let instead of var does block of the scope of i. Now clicking on any box will display the value for i that was scoped to the loop iteration.

```
var div, container = document.getElementById('container')
for (let i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
      alert('This is box #: ' + i)
   }
  container.appendChild(div)
}
```



Output

Template Strings

Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string.

Traditional string concatenation uses plus signs or commas to compose a string using variable values and strings.

```
console.log(lastName + ", " + firstName + " " + middleName)
```

With a template, we can create one string and insert the variable values by surrounding them with ${variable}.

```
console.log(`${lastName}, ${firstName} ${middleName}`)
```

Any JavaScript that returns a value can be added to a template string between the ${ } in a template string.

Template strings honor whitespace. They will make it easier to draft up email templates, code examples, or anything that contains whitespace. Now you can have a string that spans multiple lines without breaking your code.

Example 2-1. Template Strings Honor Whitespace

```
`

  Hello ${firstName},

  Thanks for ordering ${qty} tickets to ${event}.

  Order Details
    ${firstName} ${middleName} ${lastName}
    ${qty} x $${price} = $${qty*price} to ${event}

  You can pick your tickets up at will call 30 minutes before
  the show.

  Thanks,

  ${ticketAgent}

`
```

These tabs, line breaks, spaces, and variable names can be used in an email template.

Previously, using an HTML string directly in our JavaScript code was not so easy to reason about because we'd need to run it together on one line. Now the whitespace is recognized as text, and you can insert formatted HTML that is easy to understand.

```
document.body.innerHTML = `
<section>
  <header>
      <h1>The HTML5 Blog</h1>
  </header>
  <article>
      <h2>${article.title}</h2>
      ${article.body}
```

```
  </article>
  <footer>
      <p>copyright ${new Date().getYear()} | The HTML5 Blog</p>
  </footer>
</section>
`
```

Notice that we can include variables for the page title and article text as well.

Default Parameters

Languages including C++ and Python allow developers to declare default values for function arguments. Default parameters are included in the ES6 spec, so in the event that a value is not provided for the argument, the default value will be used.

For example, we can set up default strings.

```
function logActivity(name="Shane McConkey", activity="skiing") {
  console.log( `${name} loves ${activity}` )
}
```

If no arguments are provided to the favoriteActivity function, it will run correctly using the default values. Default arguments can be any type, not just strings.

```
var defaultPerson = {
    name: {
        first: "Shane",
        last: "McConkey"
    },
    favActivity: "skiing"
}

function logActivity(p=defaultPerson) {
    console.log(`${p.name.first} loves ${p.favActivity}`)
}
```

Arrow Functions

Arrow functions are a useful new feature of ES6. With arrow functions, you can create functions without using the function keyword. You also often do not have to use the return keyword.

Example 2-2. As a Traditional Function

```
var lordify = function(firstname) {
  return `${firstname} of Canterbury`
}

console.log( lordify("Dale") )       // Dale of Canterbury
console.log( lordify("Daryle") )     // Daryle of Canterbury
```

With an arrow function, we can simplify the syntax tremendously.

Example 2-3. As an Arrow Function

```
var lordify = firstname => `${firstname} of Canterbury`
```

Semi-colons throughout this Book

Semi-colons are optional in JavaScript. Our philosophy on JavaScript is why put semi-colons in that aren't required. This book takes a minimal approach that excludes unnecessary syntax.

With an arrow, we now have an entire function declaration on one line. The function keyword is removed. We also remove return because the arrow points to what should be returned. Another benefit is that if the function only takes one argument, we can remove the parentheses around the arguments.

More than one argument should be surrounded in parentheses.

```
// Old
var lordify = function(firstName, land) {
  return `${firstName} of ${land}`
}

// New
var lordify = (firstName, land) => `${firstName} of ${land}`

console.log( lordify("Dale", "Maryland") )    // Dale of Maryland
console.log( lordify("Daryle", "Culpeper") )  // Daryle of Culpeper
```

We can keep this as a one line function because there is only one statement that needs to be returned.

More than one line needs to be surrounded with brackets.

```
// Old
var lordify = function(firstName, land) {

  if (!firstName) {
    throw new Error('A firstName is required to lordify')
  }

  if (!land) {
    throw new Error('A lord must have a land')
  }

  return `${firstName} of ${land}`
}

// New
var _lordify = (firstName, land) => {
```

```
  if (!firstName) {
    throw new Error('A firstName is required to lordify')
  }

  if (!land) {
    throw new Error('A lord must have a land')
  }

  return `${firstName} of ${land}`
}

console.log( lordify("Kelly", "Sonoma") )   // Kelly of Sonoma
console.log( lordify("Dave") )              // ! JAVASCRIPT ERROR
```

These if/else statements are surrounded with brackets but still benefit from the shorter syntax of the arrow function.

Arrow functions do not block off *this*. For example, *this* becomes something else in the setTimeout callback, not the tahoe object.

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(function() {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Cannot read property 'join' of undefined
```

This error is thrown because it's trying to use the .join method on what *this* is. In this case, it's the window object. Alternatively, we can use the arrow function to protect the scope of *this*.

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Kirkwood, Squaw, Alpine, Heavenly, Northstar
```

This works correctly and we can .join the resorts with a comma. Be careful though that you're always keeping scope in mind. Arrow functions do not block off the scope of *this*.

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: (delay=1000) => {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print(); // Cannot read property resorts of undefined
```

Chaining the print function to an arrow function means that *this* is actually the window.

To verify, let's change the console message to evaluate whether this is the window.

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print(); // true
```

It evaluates as true. To fix this, we can just use a regular function.

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print() // false
```

Transpiling ES6

Most web browsers don't support ES6, and even those that do, don't support everything. The only way to be sure that your ES6 code will work is to convert it to ES5 code before running it in the browser. This process is called transpiling. One of the most popular tools for transpiling is Babel (www.babeljs.io)

In the past, the only way to use the latest JavaScript features was to wait weeks, months, or even years until browsers supported them. Now, transpiling has made it possible to use the latest features of JavaScript right away. The transpiling step makes JavaScript similar to other lanugages. Transpiling is not compiling - our code isn't compiled to binary. Instead, it's transpiled into syntax that can be interpreted by a wider range of browsers. Also, JavaScript now has source code, meaning that there will be some files that belong to your project that don't run in the browser.

Here is some ES6 code. We have an arrow function, which we will cover in a bit, mixed with some default arguments for x and y.

Example 2-4. ES6 Code before Babel Transpiling

```
const add = (x=5, y=10) => console.log(x+y);
```

After we run the transpiler on this code, here is what the output would look like:

```
"use strict";

var add = function add() {
    var x = arguments.length <= 0 || arguments[0] === undefined ?
        5 : arguments[0];
    var y = arguments.length <= 1 || arguments[1] === undefined ?
        10 : arguments[1];
    return console.log(x + y);
};
```

The transpiler added a "use strict" declaration to run in strict mode. The variables x and y are defaulted using the arguments array, a technique you may be familiar with. The resulting JavaScript is more widely supported.

You can transpile JavaScript directly in the browser using the inline Babel transpiler. You just include the browser.js file, and any scripts with `type="text/babel"` will be converted. Even though Babel 6 is the current version of Babel, only the CDN for Babel 5 will work.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.js"> </script> <scri
</script>
```

Transpiling in the Browser

This approach means that the browser does the transpiling at runtime. This is not a good idea for production because it will slow your application down a lot. In chapter 5, we'll go over how to do this in production. For now, the CDN link will allow us to discover and use ES6 features.

You may think to yourself: "Great! When ES6 is supported by browsers, we won't have to use Babel anymore!" However, as soon as this happens, we will want to use ES7 and beyond features. Unless a tectonic shift occurs, we'll likely be using Babel in the foreseeable future.

ES6 Objects and Arrays

ES6 gives us new ways to work with objects and arrays and for scoping the variables within these data sets. These features include destructuring, object literal enhancement, and the spread operator.

Destructuring Assignment

The destructuring assignment allows you to locally scope fields within an object and to declare which values will be used.

Consider this sandwich object. It has four keys, but we only want to use the values of two. We can scope bread and meat to be used locally.

```
var sandwich =  {
      bread: "dutch crunch",
      meat: "tuna",
      cheese: "swiss",
      toppings: ["lettuce", "tomato", "mustard"]
}

var {bread, meat} = sandwich

console.log(bread, meat) // dutch crunch tuna
```

The code pulls bread and meat out of the object and creates local variables for them. Also, bread and meat variables can be changed.

```
var {bread, meat} = sandwich

bread = "garlic"
meat = "turkey"

console.log(bread) // garlic
console.log(meat) // turkey
```

```
console.log(sandwich.bread, sandwich.meat) // dutch crunch tuna
```

We can also destructure incoming function arguments. Consider this function that would log a person's name as a Lord.

```
var lordify = regularPerson => {
  console.log(`${regularPerson.firstname} of Canterbury`)
}

var regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
}

lordify(regularPerson)       // Bill of Canterbury
```

Instead of using dot notation syntax to dig into objects, we can destructure the values that you need out of regularPerson.

```
var lordify = ({firstname}) => {
  console.log(`${firstname} of canterbury`)
}

lordify(regularPerson)      // Bill of Canterbury
```

Destructuring is also more declarative, meaning that our code is more descriptive about what we are trying to accomplish. By destructuring firstname, we declare that we will only use the firstname variable. More on declarative programming in the next chapter.

Values can also be destructured from arrays. Imagine that we wanted to assign the first value of an array to a variable name

```
var [firstResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(firstResort) // Kirkwood
```

You can also pass over unnecessary values with list matching using commas. List matching occurs when commas take the place of elements that should be skipped. With the same array, we can access the last value by replacing the first two values with commas.

```
var [,,thirdResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(thirdResort) // Alpine
```

Later in the section, we'll take this example a step further by combining array destructuring and the spread operator.

Object Literal Enhancement

Object literal enhancement is the opposite of destructuring. It is the process of restructuring or putting back together. With object literal enhancement, we grab variables from scope and turn them into an object.

```
var name = "Tallac"
var elevation = 9738

var funHike = {name,elevation}

console.log(funHike) // {name: "Tallac", elevation: 9738}
```

Name and elevation are now keys of the funHike object.

We can also create object methods with object literal enhancement or restructuring.

```
var name = "Tallac"
var elevation = 9738
var print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`)
}

var funHike = {name,elevation,print}

funHike.print()     // Mt. Tallac is 9738 feet tall
```

Notice we use *this* to access the object keys.

When defining object methods, it is no longer necessary to use the function keyword.

Example 2-6. Old vs. New: Object Syntax

```
OLD
var skier = {
    name: name,
    sound: sound,
    powderYell: function() {
        var yell = this.sound.toUpperCase()
        console.log(`${yell} ${yell} ${yell}!!!`)
    },
    speed: function(mph) {
        this.speed = mph
        console.log('speed:', mph)
    }
}

NEW
const skier = {
    name,
    sound,
    powderYell() {
        let yell = this.sound.toUpperCase()
```

```
        console.log(`${yell} ${yell} ${yell}!!!`)
    },
    speed(mph) {
        this.speed = mph
        console.log('speed:', mph)
    }
}
```

Object literal enhancement allows us to pull global variables into objects and reduces typing by making the function keyword unnecessary.

Spread Operator

The spread operator is three dots (...) that perform several different tasks. First, the spread operator allows us to combine the contents of arrays. For example, if we had two arrays, we could make a third array that combines the two arrays into one.

```
var peaks = ["Tallac", "Ralston", "Rose"]
var canyons = ["Ward", "Blackwood"]
var tahoe = [...peaks, ...canyons]

console.log(tahoe.join(', '))  // Tallac, Ralston, Rose, Ward, Blackwood
```

All of the items from peaks and canyons are pushed into a new array called tahoe.

Let's take a look at how the spread operator can help us deal with a problem. Using the peaks array from the previous sample, let's imagine that we wanted to grab the last item from the array rather than the first. We can use the .reverse() method to reverse the array in combination with array destructuring.

```
var peaks = ["Tallac", "Ralston", "Rose"]
var [last] = peaks.reverse()

console.log(last) // Rose
console.log(peaks.join(', '))  // Rose, Ralston, Tallac
```

Look what happens though. The reverse function has actually altered or mutated the array. In a world with the spread operator, we don't have to mutate the original array, we can create a copy of the array and then reverse it.

```
var peaks = ["Tallac", "Ralston", "Rose"]
var [last] = [...peaks].reverse()

console.log(last) // Rose
console.log(peaks.join(', '))  // Tallac, Ralston, Rose
```

Since we use the spread operator to copy the array, the peaks array is still intact and can be used later in its original form.

The spread operator can also be used to get some, or the rest, of the items in the array.

```
var lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"]

var [first, ...rest] = lakes

console.log(rest.join(", ")) // "Marlette, Fallen Leaf, Cascade"
```

We can also use the spread operator to collect function arguments as an array. We will build a function that takes in *n* number of arguments using the spread operator, and then uses those arguments to print some console messages.

```
function directions(...args) {
  var [start, ...remaining] = args
  var [finish, ...stops] = remaining.reverse()

  console.log(`drive through ${args.length} towns`)
  console.log(`start in ${start}`)
  console.log(`the destination is ${finish}`)
  console.log(`stopping ${stops.length} times in between`)
}

directions(
    "Truckee",
    "Tahoe City",
    "Sunnyside",
    "Homewood",
    "Tahoma"
)
```

The directions function takes in the arguments using the spread operator. The first argument is assigned to the start variable. The last argument is assigned to a finish variable using array.reverse(). We then use the length of the arguments array to display how many towns we're going through. The number of stops is the length of the arguments array minus the finish stop. This provides incredible flexibility because we could use the directions function to handle any number of stops.

The spread operator can also be used for objects. This is a stage-2 proposal[3] in the current ES2017 pending specification. Using the spread operator with objects is similar. In this example, we'll use it the same way we combined two arrays into a third array, but instead of arrays, we'll use objects.

```
var morning = {
  breakfast: "oatmeal",
  lunch: "peanut butter and jelly"
}

var dinner = "mac and cheese"

var backpackingMeals = {
```

3 Spread Operator, https://github.com/tc39/proposals

```
  ...morning,
  dinner
}

console.log(backpackingMeals) // {breakfast: "oatmeal",
                                  lunch: "peanut butter and jelly",
                                  dinner: "mac and cheese"}
```

Module Imports and Exports

In the earlier days of JavaScript, we would write a script that had many different functions

Promises

Promises give us a way to make sense out of asynchronous behavior. When making an asynchronous request, one of two things can happen: everything goes as we hope or there's an error. This can happen in a number of different ways. For example, we could try several ways to obtain the data to reach success. We also could receive multiple types of errors. Promises give us a way to simplify back to a simple pass or fail.

We'll build a promise that handles all of the different ways you can win or lose craps. If you've never played the game craps, that's ok. The code sample will teach it to you. The goal here is to wrangle the multiple different possible results into pass or fail.

First, we'll define all of the ways to win and lose craps. On the first roll when the point is not yet set, we win by rolling a 7 or an 11. We'll lose by rolling a 2 or a 3. We set a point by rolling any other number.

On each additional roll when the point is set, we win by hitting the point or rolling the same number again. We lose by rolling a 7. If we roll any other number, nothing happens. We roll again.

After each roll, the game is either over because you won or lost, or you have to roll again. When the game is over, this function logs whether you've won or lost.

```
const gameOver = result =>
    console.log(`Game Over - ${result}`)
```

If you are still rolling, we'll use another function to tell you what happened and what the current point is.

```
const stillRolling = (message, currentPoint) =>
    console.log(`${message} - try again for ${currentPoint}`)
```

If we had a function called craps that returned a promise, we could send the function that defines what to do when the game is over, and the function that defines what to do if we are still rolling. The craps promise will figure out which one to use.

We then can call the function that represents the first roll because we do not send a point.

```
craps(7).then(gameOver, stillRolling)
craps(2).then(gameOver, stillRolling)
craps(8).then(gameOver, stillRolling)
```

Then we can send with an additional argument to represent that the point has been set.

```
craps(5,8).then(gameOver, stillRolling)
craps(7,8).then(gameOver, stillRolling)
craps(8,8).then(gameOver, stillRolling)
```

That second argument represents where the point is set: 8.

The craps function takes the roll and the point as arguments and returns a promise object. Promise objects have a then function. Promises send the promise constructor a callback. With this callback, we will check the roll and navigate the complexities of all the different ways that craps can be won or lost.

If the game is over, gameOver is invoked with the results. If the game is still going, roll again is invoked with a message.

```
const craps = (roll, point) => new Promise((gameOver, rollAgain) => {

  // If roll is not sent as a number between 2 and 12, rollAgain
  if (!roll || typeof roll !== "number"  || roll < 2 || roll > 12) {
    rollAgain("to roll a number")

  // If a point is not set, then this must be the first roll, the come out roll
  } else if (!point) {

    // If you roll a 7 or 11 during the first role, gameOver, you loose
    if (roll === 7 || roll === 11) {
      gameOver("You win by natural")

    // If you roll a 2 or a 3, gameOver, you win
    } else if (roll === 2 || roll === 3) {
      gameOver("You lose, crapped out")

    // Otherwise the point is set, rollAgain
    } else {
      rollAgain(roll)
    }

  // It's not the first roll, and you rolled the point, gameOver. You win
  } else if (roll === point) {
```

```
      gameOver("You win, you hit the point!")

    // It's not the first roll
    } else {

      // And you rolled a 7, gameOver you loose
      if (roll === 7) {
        gameOver("You lose, craps")

      // Otherwise you missed, try again to hit the point
      } else {
        rollAgain(point)
      }
    }

})
```

There are many different outcomes for any roll. The promise defines all of them. We will then use the promise by using the .then function.

We haven't yet written a function called end that will log the outcome when the game is over.

```
const end = result =>
    console.log(`Game Over - ${result}`)
```

This just logs the outcome to the console. We also need to write the stillRolling function.

```
const stillRolling = point =>
    console.log(`The point is ${point}, try again`)
```

The game can be played using craps.then.

```
craps("foo").then(end, stillRolling) // The point is to roll a number, try again
craps(7).then(end, stillRolling) // Game Over - You win by natural
craps(2).then(end, stillRolling) // Game Over - You lose, crapped out
craps(8).then(end, stillRolling) // The point is 8, try again

craps(5,8).then(end, stillRolling) // The point is 8, try again
craps(7,8).then(end, stillRolling) // Game over - You lose, craps
craps(8,8).then(end, stillRolling) // Game over - You win, you hit the point
```

The promise can help us deal with the many different outcomes successfully. The issue here is that we're dealing with synchronous data. More often than not, promises are used with asynchronous data.

Let's create an asynchronous promise for loading data from the randomuser.me API. This API has information like email, name, phone number, location, etc. for fake members and is great to use as dummy data.

The getFakeMembers function returns a new promise. The promise makes a request to the API. If the promise is successful, the data will load. If the promise is unsuccessful, an error will occur.

```
const getFakeMembers = count => new Promise((resolves, rejects) => {
  const api = `http://api.randomuser.me/?nat=US&results=${count}`
  const request = new XMLHttpRequest()
  request.open('GET', api)
  request.onload = () =>
       (request.status === 200) ?
        resolves(JSON.parse(request.response).results) :
        reject(Error(request.statusText))
  request.onerror = (err) => rejects(err)
  request.send()
})
```

With that, the promise has been created, but it hasn't been used yet. We'll use it with a simple console log for now, but we'll use it in a larger project during Chapter 11.

We can use the promise by calling the getFakeMembers function and passing in the number of members who should be loaded. The then function can be chained on to do something once the promise has been fulfilled. This is called composition. We'll also use an additional callback that handles errors.

```
getFakeMembers(5).then(
  members => console.log(members),
  err => console.error(
      new Error("cannot load members from randomuser.me"))
)
```

Promises make dealing with asynchronous requests easier which is good because we have to deal with a lot of asynchronous data in JavaScript. You'll also see promises used heavily in Node.js, so a solid understanding of promises is essential for the modern JavaScript engineer.

Classes

Previously in JavaScript, there were no official classes. Types were defined by functions. We had to create a function and then define methods on the function object using the prototype.

```
function Vacation(destination, length) {
  this.destination = destination
  this.length = length
}

Vacation.prototype.print = function() {
    console.log(this.destination + " | " + this.length + " days")
}
```

```
var maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7
```

If you were used to classical object orientation, this probably made you mad.

Now ES6 introduces class declaration, but JavaScript still works the same way. Functions are objects, and inheritance is handled through the prototype, but this syntax makes more sense if you come from classical object orientation.

Capitalization Conventions

The rule of thumb with capitalization is that all types should be capitalized. Due to that, we will capitalize all class names.

```
class Vacation {

  constructor(destination, length) {
    this.destination = destination
    this.length = length
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`)
  }

}
```

Once we've created the class, we'll create a new instance of the class using the new keyword. Then you can call the custom method on the class.

```
const trip = new Vacation("Santiago, Chile", 7);

console.log(trip.printDetails()); // Chile will take 7 days.
```

Now that a class object has been created, you can use it as many times as you'd like to create new vacation instances. Classes can also be extended. When a class is extended, the subclass inherits the properties and methods of the super class. These properties and methods can be manipulated from here, but as a default, all will be inherited.

You can use vacation as an abstract class to create different types of vacations. For instance, an Expedition can extend the vacation class to include gear.

```
class Expedition extends Vacation {

  constructor(destination, length, gear) {
   super(destination, length)
   this.gear = gear
  }
```

```
  print() {
    super.print()
    console.log(`bring your ${this.gear.join(" and your ")}`)
  }
}
```

That's simple inheritance: the subclass inherits the properties of the super class. By calling the printDetails method of Vacation, we can append some new content onto what is printed in the printDetails method of Expedition. Creating a new instance works the exact same way: create a variable and use the new keyword.

```
const trip = new Expedition("Mt. Whitney", 3,
                    ["sunglasses", "prayer flags", "camera"])

trip.print() // bring your sunglasses and your prayer flags and your camera
```

Classes and Prototypal Inheritance

Using a class still means that you are using JavaScript's prototypal inheritance. Log Vacation.prototype, and you'll notice the constructor and printDetails methods on the prototype.

```
console.log(Vacation.prototype)
```

We will use classes a bit in this book, but we're focusing on the functional paradigm. Classes have other features like getters, setters, and static methods, but this book favors functional techniques over object-oriented techniques. The reason we're introducing these is because we'll use them later when creating React components.

JavaScript is indeed moving quickly and adapting to the increasing demands that engineers are placing on the language. Browsers are quickly implementing the features of ES6 and beyond, so it's a good idea to use these features now without hesitation.